Clean R Code cheatsheet

Clean code in R workshop, eRum 2018

Ildi Czeller (@czeildi) and Jenő Pál (@paljenczy)

May 2018

*"One difference between a smart programmer and a professional programmer is that the professional understands that **clarity is king**. Professionals use their powers for good and write code that others can understand."* (Robert C. Martin: Clean Code)

1. Use meaningful names

Use intent-revealing names

The name should tell you the purpose why an object exists, what it does, how it is used. You don't need a comment to know all these about an object.

```
nd <- 3  # number of days
```

```
number_of_days <- 3
```

Use names that you can pronounce

```
rows_w_miss_val <- df[!complete.cases(df), ]
```

```
rows_with_missing_values <- df[!complete.cases(df), ]
```

Use names that are easy to distinguish

Don't use number suffixes to distinguish variables.

```
x1 <- 72.3
x2 <- 34.5
```

```
latitude <- 72.3
longitude <- 34.5
```

Avoid having close but not the same concepts.

```
client_info <- list("id" = 123, "name" = "Foo Ltd.")
client_data <- list("id" = 123, "number_of_clients" = 34353)
```

```
client <- list("id" = 123, "name" = "Foo Ltd.", "number_of_client
s" = 34353)
```

Use one word for one concept

`get`, `retrieve`, `fetch` are synonyms. Pick one if two functions perform the same action.

```
get_client_data <- function(...)
fetch_location_data <- function(...)
```

```
get_client_data <- function(...)
get_location_data <- function(...)
```

Use verbs to name functions

Functions *do* something with inputs. As such, choose a name that reflects what it *does*.

```
client_data <- function(...)
```

```
get_client_data <- function(...)
```

Do not overwrite variables

Otherwise

- it is more difficult to inspect intermediate results
- it may happen that if you run the same code block twice you will get different results

```
customers <- delete_rows_with_missing_values(customers)
```

```
complete_customers <- delete_rows_with_missing_values(customers)
```

Choose names that do not conflict with base functions or keywords

Find out via calling the Help on the name: `?[name that you want to use]`. Example: `?data`.

```
data <- read_csv("customers.csv")
```

```
customers <- read_csv("customers.csv")
```

Do not use noise words

Avoid using name elements that show type or contain redundant information.

```
dt_customers <- read_csv("customers.csv")
locations_info <- read_csv("locations.csv")
```

```
customers <- read_csv("customers.csv")
locations <- read_csv("locations.csv")
```

Avoid magic numbers

Use named variables instead.

```
cities %>% filter(city_population > 10000)
```

```
relevant_city_size <- 10000
cities %>% filter(city_population > relevant_city_size)
```

2. Functions

Don't repeat yourself (DRY)

Instead, write a function that encapsulates the functionality in one place.

```
customers[!complete.cases(customers), ]
locations[!complete.cases(locations), ]
```

```
get_rows_with_missing_data <- function(df) {
  df[!complete.cases(df), ]
}

get_rows_with_missing_data(customers)
get_rows_with_missing_data(locations)
```

A function should do precisely one thing

"One thing": things only one level below the abstraction level of the function name. A rule of thumb: you can formulate what the function does in a short English paragraph.

If it does more than one things, break it into pieces.

```
glimpse_largest_cities <- function(home_cities, countries) {
  home_cities %>%
    group_by(country, city) %>%
    summarize(num_contact = sum(num_contact)) %>%
    arrange(desc(num_contact)) %>%
    head()
}
```

```
glimpse_largest_cities <- function(home_cities, countries) {
  home_cities %>%
    summarize_city_population() %>%
    filter_largest_units()
}

summarize_city_population <- function(cities) {
  cities %>%
    group_by(country, city) %>%
    summarize(num_contact = sum(num_contact))
}

filter_largest_units <- function(contacts_in_units) {
  contacts_in_units %>%
    arrange(desc(num_contact)) %>%
    head()
}
```

Extract code to function to express intent

Even if you don't plan to reuse the code in more than one places.

```
home_cities %>%
  group_by(country_code, city) %>%
  summarize(num_coord_per_city = n_distinct(long, lat)) %>%
  ungroup() %>%
  filter(num_coord_per_city > 1)
```

```
get_cities_with_multiple_coordinates <- function(home_cities) {
  home_cities %>%
    group_by(country_code, city) %>%
    summarize(num_coord_per_city = n_distinct(long, lat)) %>%
    ungroup() %>%
    filter(num_coord_per_city > 1)
}

get_cities_with_multiple_coordinates(home_cities)
```

Avoid too many parameters (> 3)

If you write a function with too many parameters, write two (or more) functions instead.

```
plot_share_of_cities <- function(cities, variable, share_at_least,
 add_legend) { ... }
```

```
calculate_share_of_cities <- function(cities, variable) { ... }

keep_at_least <- function(df, at_least) { ... }

plot_cities <- function(cities, add_legend) { ... }
```

Pass all parameters to a function as arguments

This makes functions self-contained.

```
all_clients <- fread("clients.csv")

get_biggest_clients <- functions(top_n) {
  all_clients %>% arrange(-num_customers) %>% head(top_n)
}

get_biggest_clients(10)
```

```
all_clients <- fread("clients.csv")

get_biggest_clients <- functions(clients, top_n) {
  clients %>% arrange(-num_customers) %>% head(top_n)
}

get_biggest_clients(all_clients, 10)
```

Very rare exception: global constants with a naming convention that is easy to follow (e.g. `ALL_CAPITAL_LETTERS`).

Clearly separate functions with side effects and functions with a return value

Side effects mean that things happen silently: a plot gets saved, a variable's value changes globally, etc. Avoid them or make them explicit.

```
plot_population <- function(country_population) {
  plot <- ggplot(country_population) +
    geom_bar(aes(x = country, y = population))
  ggsave(plot)
  plot
}
```

```
plot_population <- function(country_population) {
  plot <- ggplot(country_population) +
    geom_bar(aes(x = country, y = population))
  plot
}

save_population_plot <- function(country_population) {
  ggsave(plot_population(country_population))
}
```

Organize your functions from top to down in abstraction levels

Main functions should come first, lower level functions that they use come below them. This aids the natural order of reading and understanding.

```
get_biggest_cities <- function(city_data) { ... }

plot_cities <- function(cities) { ... }

plot_biggest_cities <- function(city_data) {
  city_data %>%
    get_biggest_cities() %>%
    plot_cities()
}
```

```
plot_biggest_cities <- function(city_data) {
  city_data %>%
    get_biggest_cities() %>%
    plot_cities()
}

get_biggest_cities <- function(city_data) { ... }
plot_cities <- function(cities) { ... }
```

3. Comments

Explain with the code itself rather than with comments

```
# calculate customer lifetime value
c_ltv <- calc_cust_LTV(cust_data)
```

```
customer_lifetime_value <- calculate_lifetime_value(customer)
```

4. General refactoring tips

First write a working code, then make it cleaner (= refactor)

Always check that after the refactor functionality did not change

Restarting the session and running your analysis from the beginning is a good practice.

Boy scout rule: if you modify something, think about leaving it a bit cleaner than it was

If you have to touch a piece of code for any reason, consider refactoring it as well. You may have better sense/ideas of your code later than writing it first even if you struggled to write clean code in the first place.

Refactoring is not writing it from scratch again

Small refactors are more effective than complete rewrites. Guarantees that your code works the same as before and that you can move on quickly.

Code should be readable also to people not familiar with R

You don't have to overdo refactoring (don't create a new function for `dplyr::filter`), however, hide cryptic parts to named piece of code (example: `apply(dt, 2, fun)`).

Scripts should be self-contained

Do not require the manual sourcing of another script or libraries, everything that should happen to run the script should be contained in the script.

Remove dead functions, do not leave commented-out code

If you don't use a function anywhere, delete it. For any piece of code, good sign of it is that you have already commented it out - get rid of it.

Useful RStudio keyboard shortcuts

Description	Windows and Linux	Mac
Insert assignment operator	Alt+-	Option + -
Run current line/selection	Ctrl+Enter	Command+Enter
Show source code for function at cursor	F2	F2
Goto File/Function	Ctrl+.	Ctrl+.
Extract function from selection	Ctrl+Alt+X	Command+Option+X
Restart R Session	Ctrl+Shift+F10	Command+Shift+F10
Reindent lines	Ctrl+I	Command+I

Literature

Robert C. Martin: *Clean code - A Handbook of Agile Software Craftsmanship.* Prentice Hall, 2009.